Programmation en assembleur

Architecture des Ordinateurs
Module M14 Semestre 4

Printemps 2008

Coordinateur du module M14: Younès EL Amrani

Formats d'opérandes en assembleur IA32

Modes d'adressage

Type	Forme	Valeur d'opérandes	Nom
Immédiate	Imm= 12q,10,0xA,Ah,$0A,1010b	Imm (val de Imm)	Immédiate
Registre	REG (REG = EAX , ESP ...)	Contenu de REG (val du registre)	Registre
Mémoire	[Imm]	Contenu de la Mémoire à l'@ Imm	Absolu
Mémoire	[REG] (REG = EAX, ESP...)	Contenu de la Mémoire à l'@ REG	Indirect
Mémoire	[REGb + Imm])	Mémoire (REGb+Imm)	Base + déplacement
Mémoire	[REGb + REGi]	Mémoire (REGb + REGi)	Indexé
Mémoire	[Imm + REGb + REGi]	Mémoire(Imm + REGb + REGi)	Indexé
Mémoire	[REGi * s]	Mémoire (REGi* s)	Indexé, pondéré
Mémoire	[Imm+ REGi * s]	Mémoire (Imm + (REGi* s))	Indexé, pondéré
Mémoire	[REGb+ REGi * s]	Mémoire (REGb + (REGi* s))	Indexé, pondéré
Mémoire	[Imm + REGb + REGi * s]	Mémoire (Imm+REGb+(REGi* s))	Indexé, pondéré

Mémoire

Adresse	Valeur
Ox100	OxFF
0x104	OxAB
0x108	Ox13
0x10C	Ox11

Registres

Registre	Valeur
EAX	Ox100
ECX	Ox1
EDX	Ox3

Que valent les opérandes suivantes ?

EAX	
[Ox104]	
0x108	
[EAX]	
[EAX+4]	
[9+EAX+EDX]	
[260+ECX+EDX]	
[0xFC+ECX*4]	
[EAX+EDX*4]	

Exemples:

Soit la mémoire
Et les registres
Suivants:

Adresse	Valeur
Ox100	OxFF
0x104	OxAB
0x108	Ox13
0x10C	Ox11

Registre	Valeur
EAX	Ox100
ECX	Ox1
EDX	Ox3

Que valent les opérandes suivantes ?

% eax	Ox100	
[Ox104]	OxAB	
0x108	Ox108	
[EAX]	OxFF	
[EAX+4]	OxAB	
[9+EAX+EDX]	Ox11	9+0x100+0x3=0x10C
[260+ECX+EDX]	Ox13	260=16*16+4=0x104
[0xFC+ECX*4]	OxFF	0xFC+0x1*4 = 0x100
[EAX+EDX*4]	0x11	0x100+0x3*4=0x10C

Principaux registres du processeur

1. Le compteur de programme EIP, Ce compteur indique (contient) l'adresse de la prochaine instruction à exécuter. Il ne peut être manipulé directement par programme.
2. Le fichier des registres d'entiers: il contient huit registres. Chaque registre peut contenir une valeur de 32 bits pour une architecture 32 bits comme IA32. Ces registres sont:
    i. EAX , EBX , ECX , EDX (généraux)
    ii. ESI , EDI (Utilisés par les chaînes)
    iii. ESP , EBP (Utilisés par la pile)

Typage niveau assembleur

En assembleur on ne fait pas de distinction entre

(1) Les entiers signés et les entiers non signés

(2) Les adresses (pointeurs) entres elles

(3) Entre les adresses et les entiers

(4) Entre les entiers et les caractères

(5)Entre structures, unions, classes, tableaux, etc

Les entiers et les pointeurs: même chose!

Typage niveau assembleur

QUESTION: Mais alors, qui effectue le typage de programmes en assembleur ??

RÉPONSE: C'est le programmeur qui assure le typage... DANS SA TÊTE !!!

Le typage est effectué par le programmeur

Structures de données coté assembleur

Les tableaux, les structures et les unions sont perçus pareillement au niveau de l'assembleur.

Les structures de données sont perçues comme

« des *collections d'octets contigüs* (côte à côte)»

Tableaux ≡ structures ≡ unions ≡ " collections d'octets contigüs"

Les instructions = opération-code + opérandes: OPCODE *(op)**

- Une instruction à un code d'opération noté opcode + une ou plusieurs opérandes.
- Les opérandes (si nécessaires) contiennent
  - les valeurs qui constituent les données de l'opération à effectuer et / ou
  - l' adresse de la destination du résultat.
- Syntaxe: Les « **( )** » signifient optionnel. Le « ***** » signifie 0 ou plusieurs. Le « **|** » signifie ou.

(label:) OPCODE (operande)* ; commentaire

Les instructions = opération-code + opérandes: OPCODE *(op)**

(label:) OPCODE (operande)* ; commentaire

- **<u>Exemples</u>:**

Ecrit la valeur décimale 1 dans le registre EAX

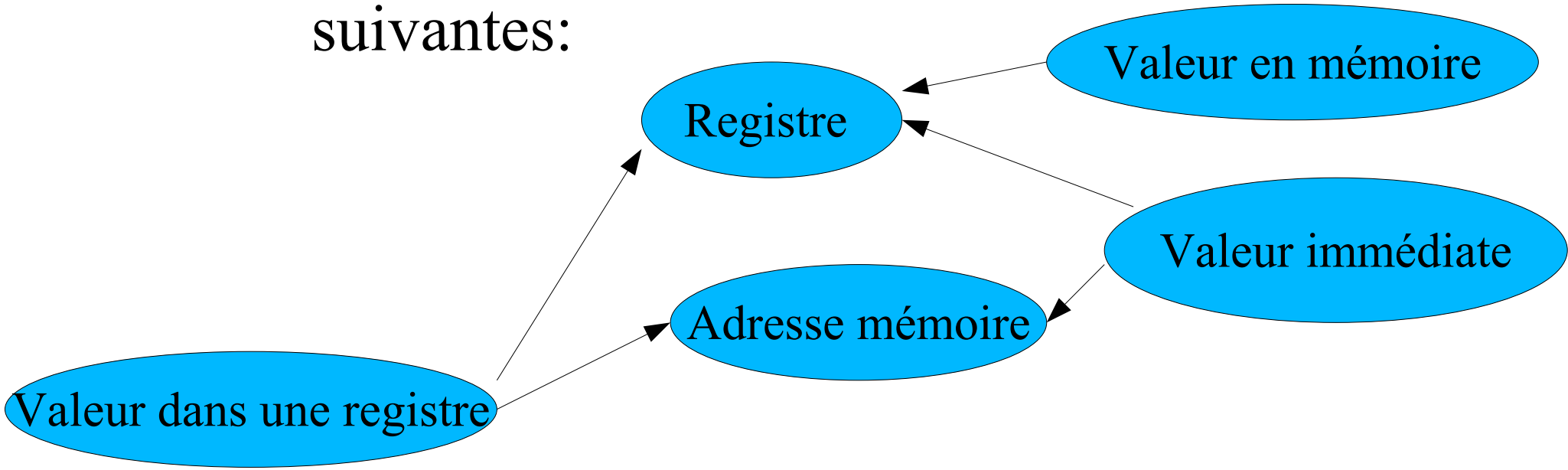
```
mov EAX , 1  ; écrit 1 dans eax
```

Lit la valeur à l'adresse EBP + 8 et l'écrit dans ESP

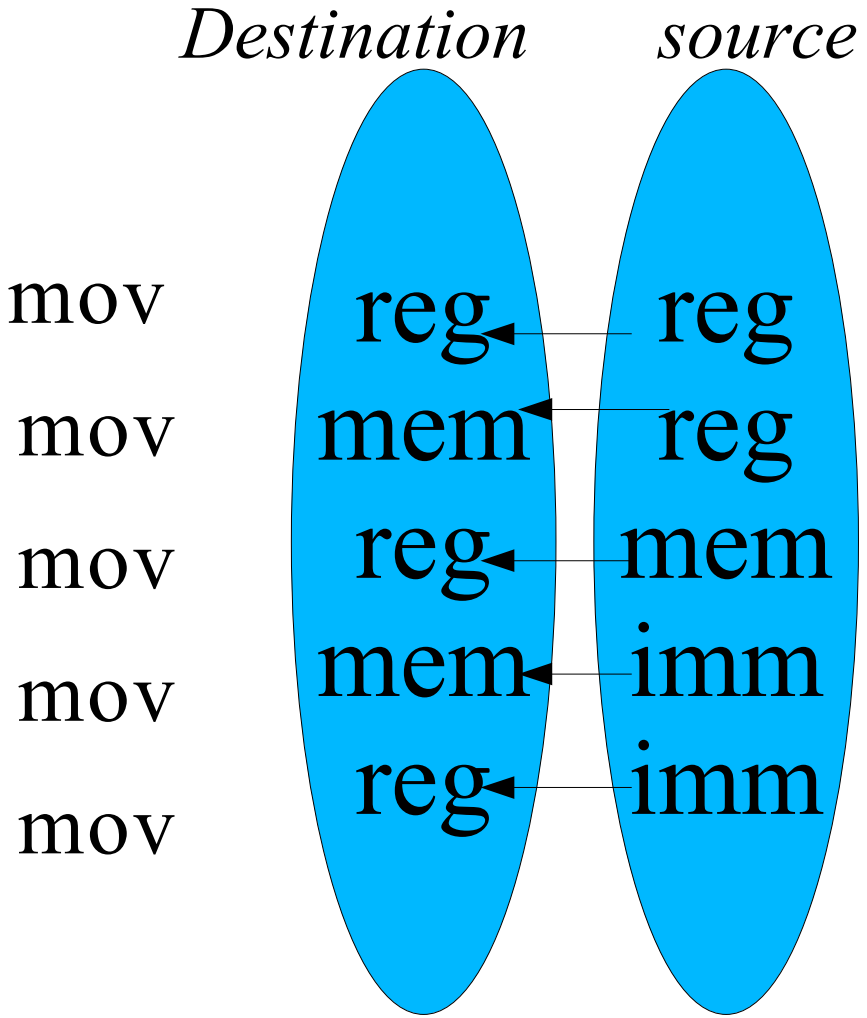
```
; lit mem(ebp+8) et l'écrit dans esp
mov ESP , [ EBP + 8 ]
```

Instructions de transfert de données

- Parmi les instruction les plus utilisées figurent celle qui déplacent les données (on dit aussi qui transfèrent les données). C'est l'instruction MOV
- Les transferts peuvent se faire dans les direction suivantes:



L'instruction move: jamais de mémoire à mémoire !



Instructions de transfert de donnée

- Une opération *MOV* ne peut avoir deux opérandes qui sont toutes les deux des adresses mémoires. Autrement dit, il n'existe pas en IA32 de transfert direct de la mémoire vers la mémoire: il faut passer par un registre.

Un mot sur l'assembleur de GCC: GAS

- Le format de l'assembleur *GNU* (noté GAS) est très différent du format standard utilisée dans la documentation d'*Intel* ainsi que d'autres compilateurs (y compris *Microsoft*) une différence majeure est dans l'inversion de l'ordre des opérandes source et destination. Ainsi que le suffixe % dont sont affublés les registres.
- Dans GAS les registres EAX, EBX, etc se notent:

%EAX

%EBX etc

GNU Assembleur Versus Assembleur INTE

- GAS documente ses propres différences avec les notations standard d'Intel.
- La commande ***info as*** dans l'environnement Linux permet d'obtenir la documentation sur GAS dans l'environnement Linux.
- Une sous-section est réservée à la comparaison de GAS avec la notation standard d'Intel.
- Dans la pratique: l'assembleur GAS est utilisé uniquement pour le code généré de gcc.

L'assembleur sous LINUX

- Observation 1:

  Il est interessant de connaître GAS pour s'inspir

  du code produit pour les programmes C et C++

- Observation 2:

  Cependant, il est plus intéressant de connaître le
  instructions de la documentation d'Intel qui
  s'imposent comme un standard.

Comparaison entre GAS et INTEL

- Différence 1

en GAS nous avons

opération source, destination

en IA32 on a

opération destination , source

- Différence 2

Pour chaque opération de GAS un caractère en suffixe indique la taille de l'opérande:

- b pour byte (1 octet)
- w pour word (16 bits) et
- L pour long words (32 bits)

Format des instructions dans un assembleur

(label:) opcode (opérandes) ; commentaire*

- Généralement on distingue deux types d'instructions:

  1- les instructions directives donnent des directives qui ne sont pas des instructions exécutable. Ex: ***extern _printf***

  2- les instructions exécutables. Ex: PI: DD 3.14 ; le réel pi

- Exemples d'instructions *directive:*

  extern _printf ; printf sera importée

  global _main ; le _main sera exporté

Allocation de données initialisées

Nombre d'octets	Mnémonique	Description	Attribut
1	D B	Définit un byte (un octet)	Byte
2	D W	Définit un mot (un " Word " en anglais)	Word
4	DD	définit un double mot (Double Word)	Double word
8	DQ	définit un mot quadruple (quadruple word)	Quad word
10	DT	Définit dix octets (dix bytes)	Ten bytes

Données initialisées en NASM

```
;;;Ce programme affiche "Hello, World"
SEGMENT .data ; données initialisées
textHello:          DB "Hello, World !" , 0
integerFormat:      DB "%d" , 0
stringFormat:       DB "%s" , 0
characterFormat:    DB "%c" , 0
charVal1:     DB  'C'
intVal1:      DD 15
intVal2:      DD 20
```

Allocation de données non initialisées

Nombre d'octets	Mnémonique	Description	Nom
1 * n	RESB n	Réserve n octet(s)	Byte
2 * n	RESW n	Réserve n mots (1 mot = 2 octets)	Word
4 * n	RESD n	Réserve n double mot(s) (1 Double mot = 4 octets)	Double word
8 * n	RESQ n	Réserve un quadruple mot (quadruple word)	Quad word
10 * n	REST n	Réserve dix octets (dix bytes)	Ten bytes

Données non-initialisées en NASM

```
;;;segment de données non initialisées
SEGMENT .bss
textHello:      RESB 15
textSalam:      RESB 5
tableauInt1:    RESD 10
tableauInt2:    RESD 15
charVal1:       RESB  1
charVal2:       RESB  1
```

Programme proprement dit

.text ;marque le début du segment de code

global _main ; *symboles exportés vers l'extérieur*

; *fonctions importées*

extern _printf , _scanf , _f1 , _f2

_main:

;;;*la directive **proc** annonce le début de la*
;;;procédure main. Cette procédure est
;;;fondamentale: elle marque le début de l'entrée
;;;du code dans le protocol C

Programme Hello World utilise la librairie C

```
_main: enter 0, 0
        pusha   ; EMPILER TOUS LES REGISTRES
        pushf   ; EMPILER LE REGISTRE D'ETAT
        mov eax , hello ; ADRESSE DU TEXTE HELLO
        push EAX        ; EMPILER ADRESSE HELLO
        push integerFormat ; EMPILER FORMAT
        call   printf           ; APPEL DE FONCTION
        pop    ecx      ; DEPILER UNE FOIS <=> ADD ESP, 4
        pop    ecx      ; dépiler une seconde fois  <=> ADD ESP, 4
```

Fin Du Programme Hello World

```
mov eax , 0   ;;; Signifie la fin de programme pour l'OS
leave         ;;; libère la zone utilisée sur la pile
ret           ;;; restaure le registre EIP
```

Les Macros en NASM

%macro nomMacro n ; n = nombre d'arguments

OPCODE1 <operandes>

...

OPCODEp <operandes>

%endmacro

Dans le corps de la macro, %1 dénote le premier argument en ligne arg1 ,..., %n référence argn

APPEL: nomMacro arg1 , ... , argn

RISC versus CISC

- En fait, avant les années 80, la notion de RISC était inexistante. La tendance était de rajouter autant d'instructions que possible au processeur.
- En fait, c'est dans les années 80, que des chercheurs d'**IBM** sous la direction de **John Cocke** se sont convaincus qu'un ensemble réduit d'instructions " rapides" valait mieux qu'un grand ensemble d'instructions parfois plus lentes.

RISC versus CISC

- Les processus RISC seraient moins chers, moins complexes et ne comporteraient que des instructions très rapides toutes codées sur un même nombre d'octets.
- Les professeurs David Patterson (ami Berkeley et John Hennessy (Stanford) sont ceux qui apposèrent les noms de CISC et RISC aux deux philosophies.

RISC	CISC
Peu d'instructions (~100)	Beaucoup d'instructions (~ 1000)
Instructions rapides: 1 cycle = 1 instruction	Des instructions parfois lentes (> 1 cycle)
Instructions toute codées sur 4 octets	Instructions codées sur 1 à 15 octets
Une base + 1 déplacement pour l'adressage	Format complexe car plusieurs formats utilisés pour l'adressage mémoire
Opérations arithmétiques et logiques sur les registres uniquement	Opération arithmétiques et logiques à la fois sur des registres et de la mémoire
Contraintes d'implémentation: séquence d'instructions interdites	Implémentation transparente
Seuls des tests, dont le résultat va dans des registres, sont utilisés lors des branchements conditionnels	Des drapeaux sont positionnés et utilisés pour lors des branchements conditionels
Utilisation uniquement des registres pour les arguments des fonctions ainsi que pour l'adresse de retour	Utilisation intensive de la pile pour les arguments et pour l'adresse de retour

RISC versus CISC

- Dans les années 80 la communauté scientifique a longuement débattu de l'avantage de l'une et l'autre philosophie.
- Dix ans plus tard, il est apparu que l'une et l'autre avait des avantages. Ainsi les processus RISC ont tendance à devenir de + en + CISC et vice-versa.
- La technologie CISC domine le marché des ordinateurs de bureau et des ordinateurs portables. La technologie RISC domine le marché des microprocesseurs embarqués.

Documentation

- Sur le site d'intel **http://download.intel.com/**

On trouve les documents suivants:

- Volume I volume 1 Basic Architecture, 1999 donne un panorama de l'architecture du point de vue d'un programmeur en assembleur.
- *download.intel.com/design/pentiumII/manuals/24319002.PDF*

- Volume II Intel Architecture Software Developer's Manual volume 2

Ensemble des Instructions Volume

Instruction Set Architecture (ISA), contenu dans le

Volume 2. Donne une description détaillée des différentes instructions disponibles sur le microprocesseur.

http: //download.intel.com/design/pentiumII/

manuals/24319102.PDF

Manuel du développeur
Volume III

Intel Architecture Software Developer's Manual

download.intel.com/design/pentiumII/manuals

/24319202.PDF

Module M14. Resp. Younès EL AMRANI.

Le pré-processor / macro-expanseur

Assembleur

Younès El Amrani

Date de création: Mai 2006

Dernière modification: mai 2010

Le pré-processeur / macro-expanseur NASM

- Toutes les macro-instructions commencent par le symbole « ***%*** »
- Le préprocesseur Nasm offre plusieurs possibilités:
  - L'assemblage conditionnel
  - L'inclusion multiple de fichiers

un fichier qui inclut un fichier qui inclut un autre fichier...

  - Deux formes de macros:
    - Les macros définies sur une seule ligne
    - Les macros définies sur plusieurs lignes

Macros définies sur une ligne

- On utilise la macro instruction *%define*
- L'utilisation de *%define* est similaire au C
- Exemple:

```
;; Soient les deux macros suivantes:
%define crtl 0x1F  &
%define param( a, b )  ( ( a ) + ( a ) * ( b ) )
;; Soit  l'instruction assembleur suivante:
mov byte [  param( 2, ebx ) ]  , crtl   'D'
;; sera expansée en:
mov byte [  ( 2 ) + ( 2 ) * ebx ) ]  ,  0x1F    &    'D'
```

Macros expansées lors de l'invocation

- Les macros sont expansées lors de leurs invocations et non lors de leurs définitions
- Exemple:

;; Soient les deux macros suivantes:

%define a(x) 1 + b(x)

*%define b(x) 2 * x*

;; Soit l'instruction assembleur suivante:

mov ax, a (8)

;; sera expansée en:

*mov ax, 1 + 2 * 8 ;; cette expansion a lieu bien que b a été définie après a*

noms de macros: minuscules ou majuscules significatives !

- Les noms de macros sont sensibles aux minuscules et majuscules
- Exemple:

;; Soient la macro

%define foo bar

;; soit l'instruction assembleur suivante:

opcode foo, Foo

;; sera expansée en:

opcode bar, Foo ;; seule foo a été expansée car Foo diffère de foo.

Cependant la définition %idefine foo bar est insensible aux majuscules / minuscules. Le « i » de idefine signifie en anglais « insensitive »

Les macros circulaires

- Le préprocesseur n'expansera une macro qu'une seule fois si la macro contient une référence à elle-même. On parle alors de macros circulaires.
- Exemple:

;; Soient la macro définition suivante:

%define a(x) 1 + a(x)

;; Soit la macro instruction suivante:

mov ax, a(3)

;; sera expansée en:

mov ax, 1 + a(3) ;; aucune autre expansion n'aura lieu

Les macros surchargées

- Il est possible de surcharger les noms de macros
- Cependant une définition de macro sans paramètres ne permet plus de définition de macros avec paramètres et vice-versa.
- Exemple:

;; Soient la macro définition suivante:

%define foo(x) 1 + x

;; Il est possible de surcharger la macro définition ci-dessus:

*%define foo(x , y) 1 + (x * y)*

;; Le macro processeur fera la différence entre les deux macros en comptant

;; les paramètres

La concaténation de chaînes dans les macros

- Il est possible de concaténer des chaînes de caractère à l'aide de l'opérateur %+
- Exemple:

;; Soient le(s) macro(s) définition(s) suivante(s):

%define _mafonction _autrefonction

%define cextern(x) _ %+ x

;; Soit la macro suivante:

cextern (mafonction)

;; Le macro processeur dans une première passe génèrera

_mafonction

;; et dans une seconde passe il génèrera

_autrefonction

Un-définir des macros

- Les macros définies par une seule ligne peuvent-être in-définies à l'aide de la commande

%undef

- Exemple:

;; Soient le(s) macro(s) définition(s) suivante(s):

%define foo bar

;;; some code dans lequel « foo » sera relplacé par « bar »

%undef foo

;; Soit l'instruction

mov eax , foo

;; Le macroprocesseur va générer

mov eax, foo ;; plus de changements car la macro foo a été undéfinie

Macros définies sur plusieurs lignes (syntaxe NASM)

- Exemple:

```
;; Soient le(s) macro(s) définition(s) suivante(s):
%macro prologueC  1  ;; le 1 signifie 1 seul argument, il sera référencé par %1
push ebp
mov  ebp , esp
sub   esp , %1
%endmacro
;; De manière générale:
%macro   <name>     <n>   ;; n est le nombre d'arguments
    <instructions>              ;;  %1 référence le 1er argument...%n réf. le nième
%endmacro      ;; pas de changements car  la macro   foo a été undéfinie
```

Macros avec des étiquettes

- Si on définit une macro qui contient une étiquette, on doit alors écrire l 'étiquette avec le préfixe %%
- Ainsi une étiquette unique sera générée à chaque appel de la macro.

Programmation assembleur : aperçu

Programmation en assembleur : NASM
Module M14 Semestre 4

Printemps 2010

Equipe pédagogique : Younès El Amrani, Abdelhakim El Imrani, Faissal Ouardi

Tout l'assembleur dans un transparent

- L'assembleur Nasm est un langage de mnémoniques
- Nasm permet de programmer la machine au niveau le plus bas.
- Nasm utilise des codes d'opérations qui activent les circuits combinatoires du processeur.
- Nasm accède aux registres de la machine. Les registres sont des circuits séquentiels.
- Nasm accède et manipule directement la mémoire en utilisant des adresses vues comme des adresses d'octets
- Nasm possède 5 types fondamentaux :
    1. byte (1 octet = 8 bits) ,
    2. word (2 octets = 16 bits)
    3. double word (4 octets = 32 bits)
    4. Quadruble word (8 octets = 64 bits)
    5. Ten Bytes (10 octets = 80 bits)

Format d'un Programme NASM

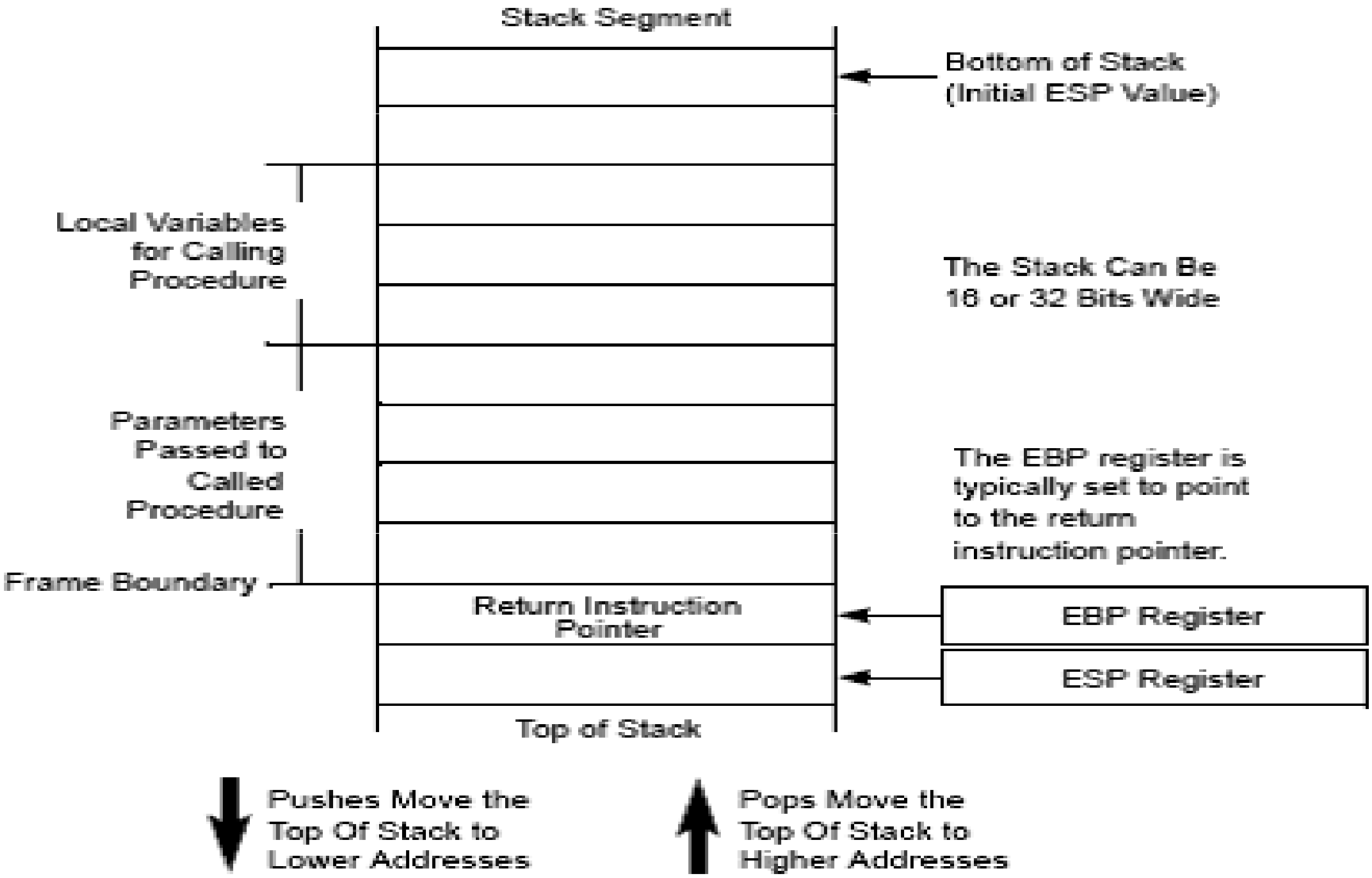
```
SEGMENT .data                          ;;; données initialisées
textHello:  DB "Hello, World !" , 0 ;;; DB signifie Define Byte
```

```
SEGMENT .bss                           ;;; les données non initialisées
textReponseHello:   RESB 15            ;;; RESB signifie Réserve Byte
```

```
SEGMENT .text                          ;;; Le code exécutable

_main:                    ;;; main est par convention l'entrée du code / le début
    ENTER 0, 0            ;;; Le prologue : on commence une fenêtre de pile
    OpCode_1   OPERANDE(S)_1      ;;; CODE OPERATION 1 suivi des opérandes ;
    ...
    OpCode_n   OPERANDE(S)_n      ;;; CODE OPERATION n suivi des opérandes ;
    LEAVE                         ;;; On referme la fenêtre de pile ;
    RET                           ;;; On restaure le compteur de programme
```

Le segment de pile



Données initialisées en NASM

```
SEGMENT .data          ;;; Le segment .data contient les données initialisées
textHello:      DB "Hello, World !" , 0   ;;; texte qui se termine par zéro
integerFormat:  DB "%d" , 0               ;;; texte format %d de printf du C
stringFormat:   DB "%s" , 0               ;;; texte format %s de printf du C
characterFormat: DB "%c" , 0              ;;; texte format %c de printf du C
charVal1:       DB  'C'                   ;;; Le caractère 'C'
intVal1:        DD 15                     ;;; L'entier 15 codé sur 32 bits
intVal2:        DD 20                     ;;; L'entier 20 codé sur 32 bits
```

Allocation de données initialisées

Nombre d'octets	Mnémonique	Description	Attribut
1	D B	Définit un byte (un octet)	Byte
2	D W	Définit un mot (un " Word " en anglais)	Word
4	DD	définit un double mot (Double Word)	Double word
8	DQ	définit un mot quadruple (quadruple word)	Quad word
10	DT	Définit dix octets (dix bytes)	Ten bytes

5.1. FUNDAMENTAL DATA TYPES

The fundamental data types of the IA are bytes, words, doublewords, and quadwords (refer to Figure 5-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), and a quadword is 8 bytes (64 bits).

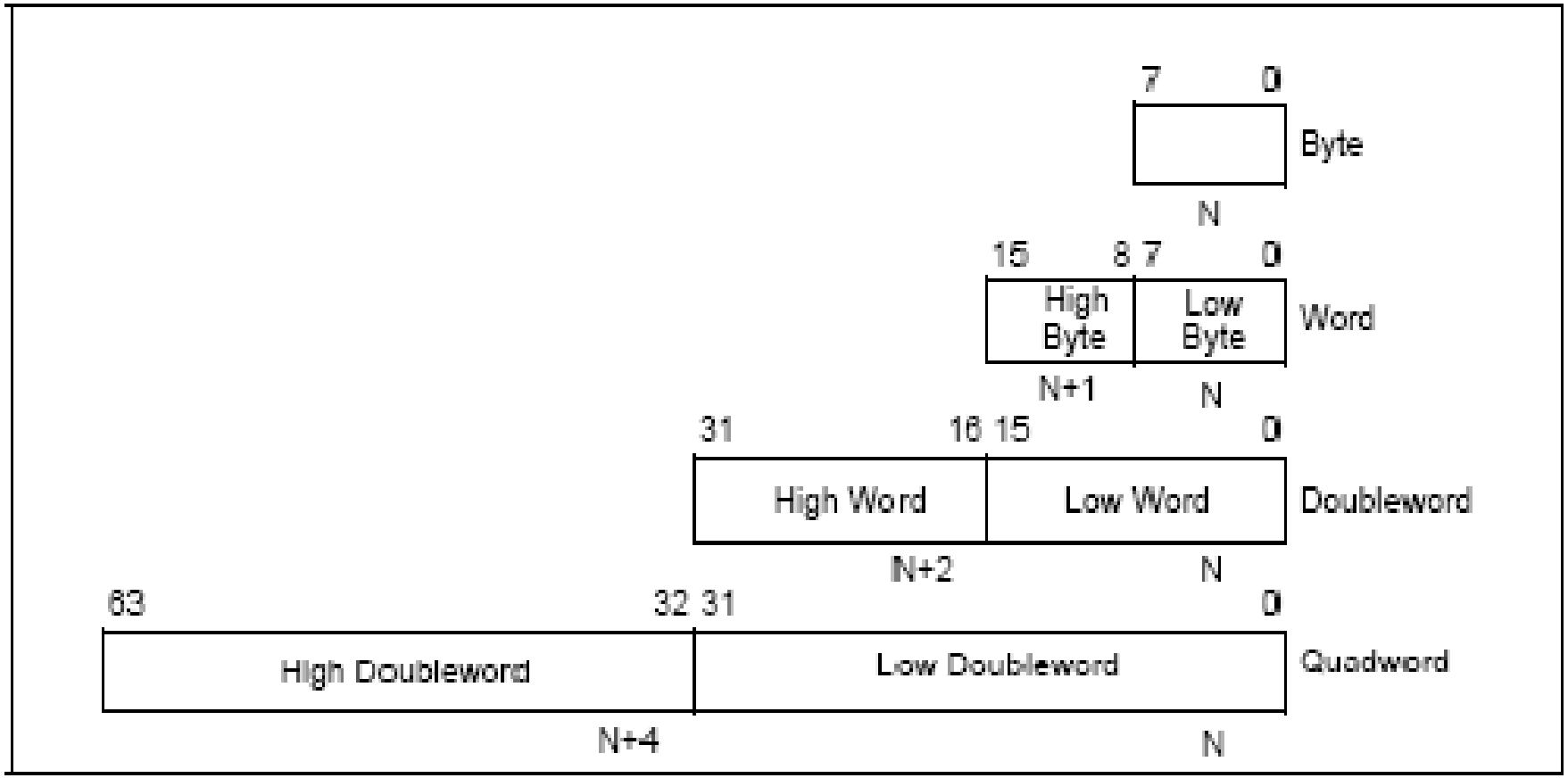
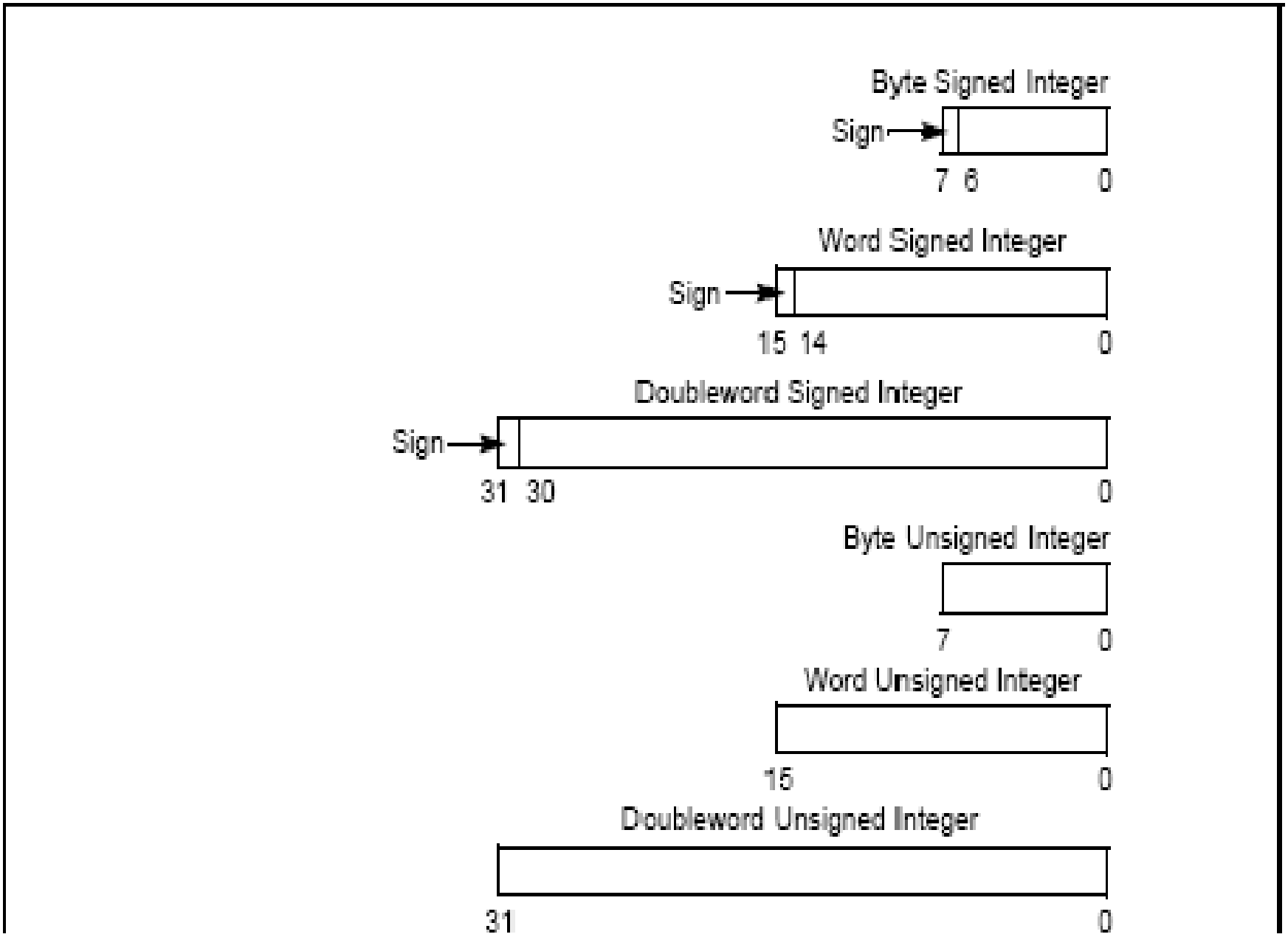


Figure 5-1. Fundamental Data Types


Byte Signed Integer
Sign
7 6 0
Word Signed Integer
Sign
15 14 0
Doubleword Signed Integer
Sign
31 30 0
Byte Unsigned Integer
7 0
Word Unsigned Integer
15 0
Doubleword Unsigned Integer
31 0

Données non-initialisées en NASM

```
SEGMENT .bss                ;;;segment de données non initialisées
    textHello:      RESB 15     ;;; Réservation de 15 Bytes (15 caractères)
    textSalam:      RESB 5      ;;; Réservation de 5 Bytes (5 caractères)
    tableauInt1:    RESD 10     ;;; Réservation de 10 entiers : c'est un tableau
    tableauInt2:    RESD 15     ;;; Réservation de 15 entiers : c'est un tableau
    charVal1:       RESB  1     ;;; Réservation de 1 seul Byte (1 seul caractère)
    charVal2:       RESB  1     ;;; Réservation de 1 seul Byte (1 seul caractère)
```

;;; DANS CE SEGMENT, LES DONNEES NE RECOIVENT PAS DE VALEUR

;;; INITIALE.

;;; UN ESPACE LEUR EST RESERVE POUR CONTENIR LA / LES VALEUR(S)

Allocation de données non initialisées

Nombre d'octets	Mnémonique	Description	Nom
1 * n	RESB n	Réserve n octet(s)	Byte
2 * n	RESW n	Réserve n mots (1 mot = 2 octets)	Word
4 * n	RESD n	Réserve n double mot(s) (1 Double mot = 4 octets)	Double word
8 * n	RESQ n	Réserve un quadruple mot (quadruple word)	Quad word
10 * n	REST n	Réserve dix octets (dix bytes)	Ten bytes

Données initialisées en NASM

```
SEGMENT .data    ;;; Le segment .data contient les données initialisées
    textHello:          DB "Hello, World !" , 0   ;;; texte qui se termine par zéro
    integerFormat:      DB "%d" , 0               ;;; texte format %d de printf du C
    stringFormat:       DB "%s" , 0               ;;; texte format %s de printf du C
    characterFormat:    DB "%c" , 0               ;;; texte format %c de printf du C
```

Programme proprement dit

```
SEGMENT .data    ;;; Le segment .data contient les données initialisées
    textHello:        DB "Hello, World !" , 0   ;;; texte : terminé par zéro
    integerFormat:    DB "%d" , 0               ;;; format %d de printf du C
    stringFormat:     DB "%s" , 0               ;;; format %s de printf du C
    characterFormat:  DB "%c" , 0               ;;; format %c de printf du C
```

```
SEGMENT .text                ;;;marque le début du segment de code
global _main                 ;;; symboles exportés vers l'extérieur
extern  _printf , _scanf     ;;; fonctions importées
```

Programme qui utilise la librairie C

```
_main:
    enter 0, 0              ;;; Ouvre une fenêtre de pile
    push textHello          ;;; ADRESSE DU TEXTE HELLO
    push stringFormat       ;;; EMPILER FORMAT
    call  _printf           ;;; APPEL DE FONCTION
    add   ESP , 8           ;;;  DEPILER 2 FOIS (ADD ESP,4)
    mov EAX , 0             ;;; Signifie fin du programme
    leave                   ;;; libère la zone utilisée sur la pile
    ret                     ;;; restaure EIP.
                            ;;; EIP signifie extended Instruction
                            ;;; Pointer
```

Quelques abbréviations

- **reg8** dénote un registre général 8 bits
- **reg16** dénote un registre général 16 bits
- **reg32** dénote un registre général de 32 bits.
- **imm** dénotes une valeur immédiate de 8, 16 ou 32 bits.
- **imm8** dénotes une valeur immédiate de 8 bits.
- **imm16** dénotes une valeur immédiate de 16 bits.
- **imm32** dénotes une valeur immédiate de 32 bits.
- **mem** dénotes une référence mémoire générique de 8, 16 ou 32 bits
- **mem8 , mem16 , mem32 , mem64** dénote une référence mémoire respectivement de 8, 16, 32 et 64 bits.
- **r/m8** dénote un registre ou une référence mémoire de 8 bits.
- **r/m16, r/m32** dénote un registre ou une référence mémoire respectivement de 16 bits et 32 bits.

PUSH : documentation NASM

- PUSH : Push Data on Stack
- PUSH reg16 ; o16 50+r [8086]
- PUSH reg32 ; o32 50+r [386]
- PUSH imm8 ; 6A ib [186]
- PUSH imm16 ; o16 68 iw [186]
- PUSH imm32 ; o32 68 id [386]

En mode 32 bits : PUSH décremente le sommet de pile (ESP) de 4 bytes, et empile la valeur donnée à [SS:ESP]

En mode 16 bits : PUSH décremente le sommet de pile (SP) de 2 bytes, et empile la valeur donnée à [SS:SP] .

LE SEGMENT .data : vue binaire !!

```
00000000 48                 H:                  DB 'H'
00000001 65                 e:                  DB 'e'
00000002 6C                 l:                  DB 'l'
00000003 64                 d:                  DB 'd'
00000004 73                 s:                  DB 's'
00000005 25                 pourcentage:        DB '%'
00000006 00                 zero:               DB 0
00000007 48656C6C6F20576F72-    textHello:      DB "Hello World" , 0
00000010 6C6400
00000013 257300             stringFormat:       DB "%s" , 0
```

LE SEGMENT .text : vue binaire !!

```
00000004    68[07000000]          push    textHello
00000009    68[13000000]          push    stringFormat
0000000E    E8(00000000)          call   _printf
00000013    81C404000000          add     ESP , 4

00000019    B800000000            mov     eax , 0
0000001E    C9                    leave
0000001F    C3                    ret
```

Principaux registres du processeur

1. Le compteur de programme EIP, Ce compteur indique (contient) l'adresse de la prochaine instruction à exécuter. Il ne peut être manipulé directement par programme.
2. Le fichier des registres d'entiers: il contient huit registres. Chaque registre peut contenir une valeur de 32 bits dans le cas d'une architecture 32 bits comme IA32. Ces registres sont:
    i. EAX , EBX , ECX , EDX (généraux)
    ii. ESI , EDI (Utilisés par les chaînes)
    iii.ESP , EBP (Utilisés par la pile)

Ajouter içi les registres de IA32 à partir de la documentation d’intel

Typage niveau assembleur

En assembleur on ne fait pas de distinction entre

(1) Les entiers signés et les entiers non signés

(2) Les adresses (pointeurs) entres elles

(3) Entre les adresses et les entiers

(4) Entre les entiers et les caractères

(5) Entre structures, unions, classes, tableaux, etc

TOUT TYPE EST UN ENSEMBLES DE UN OU PLUSIEURS OCTETS…

Typage niveau assembleur

QUESTION: Mais alors, qui effectue le typage des programmes en assembleur ??

RÉPONSE: C'est le programmeur qui assure le typage... Il n’a pas intérêt à se tromper…

Typage: dans la tête du programmeur

Structures de données coté assembleur

Les tableaux, les structures et les unions sont perçus pareillement au niveau de l'assembleur.

Les structures de données composées sont perçues comme

« des *collections d'octets contigüs* (côte à côte)»

Tableaux ≡ structures ≡ unions ≡ " collections d'octets contigüs"

instruction = opcode + opérande(s)

- Une instruction à un code d'opération noté opcode + une à plusieurs opérandes.
- Les opérandes (si nécessaires) contiennent
  - les valeurs qui constituent les données de l'opération à effectuer et / ou
  - l' adresse de la destination du résultat.
- Syntaxe:
  - Les « ( ) » signifient optionnel.
  - Le « * » signifie 0 ou plusieurs.
  - Le « | » signifie ou.

(label:) OPCODE (operande)* (; commentaire)

instruction = opcode + opérande(s)

- Syntaxe:
  - Les « **( )** » signifient optionnel.
  - Le « ***** » signifie 0 ou plusieurs.
  - Le « **|** » signifie ou.

(label:) OPCODE (operande)* (; commentaire)

Les instructions = opération-code + opérandes: OPCODE ***(op)****

(label:) OPCODE (operande*) (; commentaire)

- **Exemples:**

;;; Pour écrire la valeur décimale 1 dans le registre EAX:

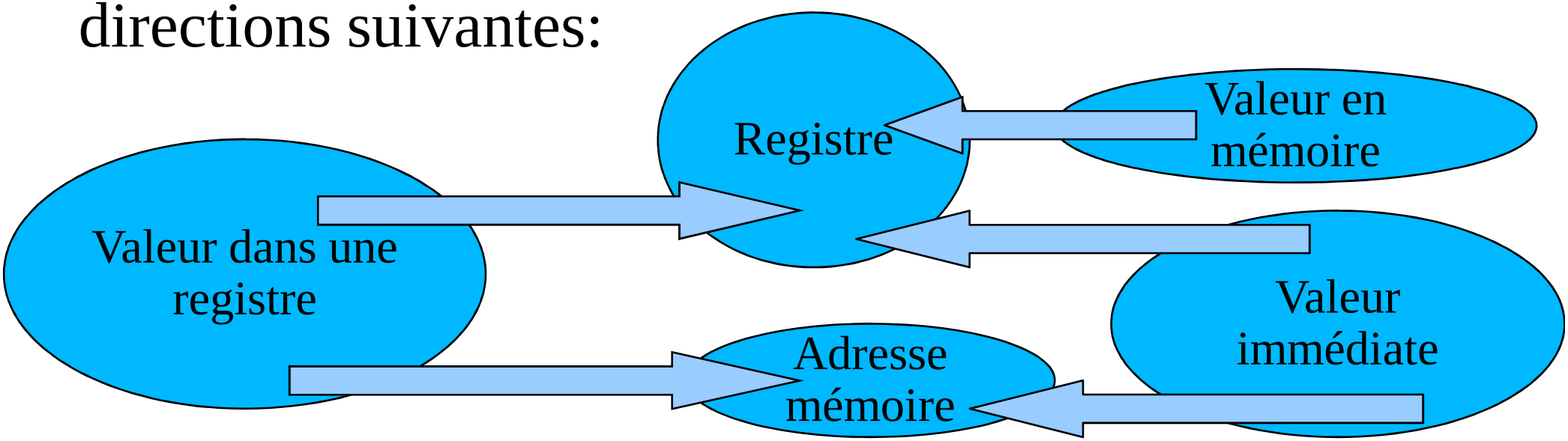
```
mov         EAX , 1              ; écrit 1 dans eax
```

;;; Pour lire Mémoire(EBP+8) et l'écrire dans ESP

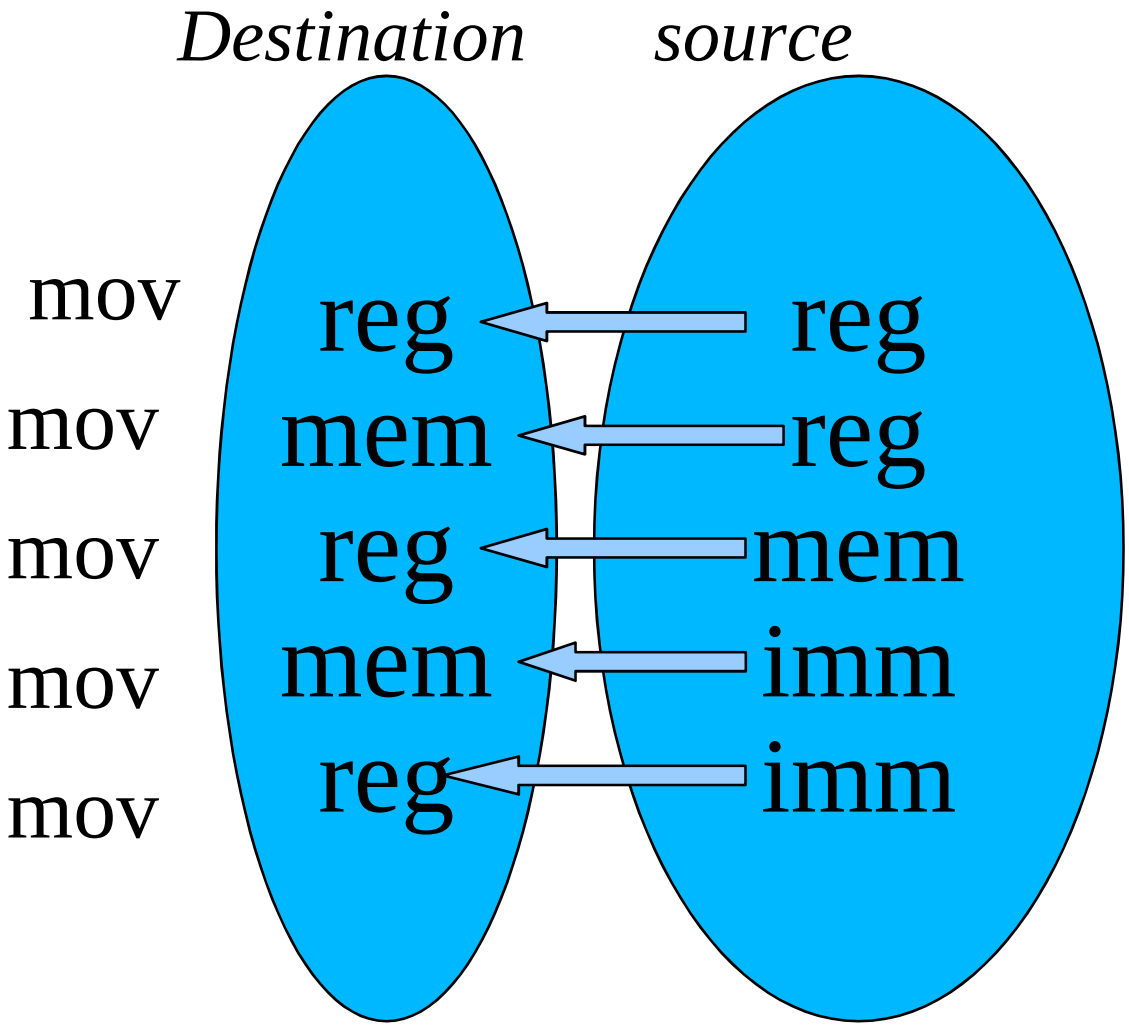
```
mov ESP , [ EBP + 8 ]
```

Instructions de transfert de données

- Parmi les instructions les plus utilisées figure celle qui déplace/transfère les données. C'est l'instruction dont le code d'opération (**opcode**) est ***MOV***. Les transferts peuvent se faire dans les directions suivantes:



L'instruction mov, jamais de mémoire à mémoire : il faut passer par un registre



Instructions de transfert de données

- Une opération ***MOV*** ne peut avoir deux opérandes qui sont toutes les deux des adresses mémoires. Autrement dit, il n'existe pas en IA32 de transfert direct de la mémoire vers la mémoire: il faut passer par un registre. Cette limitation est liée au matériel et non pas à la théorie !

Format des instructions dans un assembleur

(label:) opcode (opérandes*) (; commentaire)

➔Généralement on distingue deux types d'instructions:

1- les instructions directives donnent des directives qui ne sont pas des instructions exécutable. Ex: ***extern _printf***

2- les instructions exécutables. Ex: PI: DD 3.14 ; le réel pi

➔Exemples d'instructions *directive:*

extern _printf ; printf sera importée

global _main ; le _main sera exporté

Formats d'opérandes en assembleur IA32

Modes d'adressage

Type	Forme	Valeur d'opérandes	Nom
Immédiate	Imm= 12q,10,0xA,Ah,$0A,1010b	Valeur immédiate de Imm)	Immédiate
Registre	REG	Valeur contenue dans REG	Registre
Mémoire	[Imm]	Mémoire (Imm)	Absolu
Mémoire	[REG]	Mémoire (REG)	Indirect
Mémoire	[REGb + Imm])	Mémoire (REGb+Imm)	Base + déplacement
Mémoire	[REGb + REGi]	Mémoire (REGb + REGi)	Indexé
Mémoire	[Imm + REGb + REGi]	Mémoire (Imm + REGb + REGi)	Indexé
Mémoire	[REGi * s]	Mémoire (REGi* s)	Indexé, pondéré
Mémoire	[Imm+ REGi * s]	Mémoire (Imm + (REGi* s))	Indexé, pondéré
Mémoire	[REGb+ REGi * s]	Mémoire (REGb + (REGi* s))	Indexé, pondéré
Mémoire	[Imm + REGb + REGi * s]	Mémoire (Imm+REGb+(REGi* s))	Indexé, pondéré

Note 1: REG peutêtre un des registres EAX , EBX, ECX, EBP, ESP, EDI, ESI etc ...

Note 2: Mémoire(@) signifie contenue de la mémoire à l'adresse @

Mémoire

Adresse	**Valeur**
Ox100	OxFF
0x104	OxAB
0x108	Ox13
0x10C	Ox11

Registres

Registre	**Valeur**
EAX	Ox100
ECX	Ox1
EDX	Ox3

Que valent les opérandes suivantes ?

EAX	
[Ox104]	
0x108	
[EAX]	
[EAX+4]	
[9+EAX+EDX]	
[260+ECX+EDX]	
[0xFC+ECX*4]	
[EAX+EDX*4]	

Exemples:

Soit la mémoire
Et les registres
Suivants:

Adresse	Valeur
Ox100	OxFF
0x104	OxAB
0x108	Ox13
0x10C	Ox11

Registre	Valeur
EAX	Ox100
ECX	Ox1
EDX	Ox3

Que valent les opérandes suivantes ?

% eax	Ox100	
[Ox104]	OxAB	
0x108	Ox108	
[EAX]	OxFF	
[EAX+4]	OxAB	
[9+EAX+EDX]	Ox11	9+0x100+0x3=0x10C
[260+ECX+EDX]	Ox13	260=16*16+4=0x104
[0xFC+ECX*4]	OxFF	0xFC+0x1*4 = 0x100
[EAX+EDX*4]	0x11	0x100+0x3*4=0x10C

Code niveau machine des instructions IA32

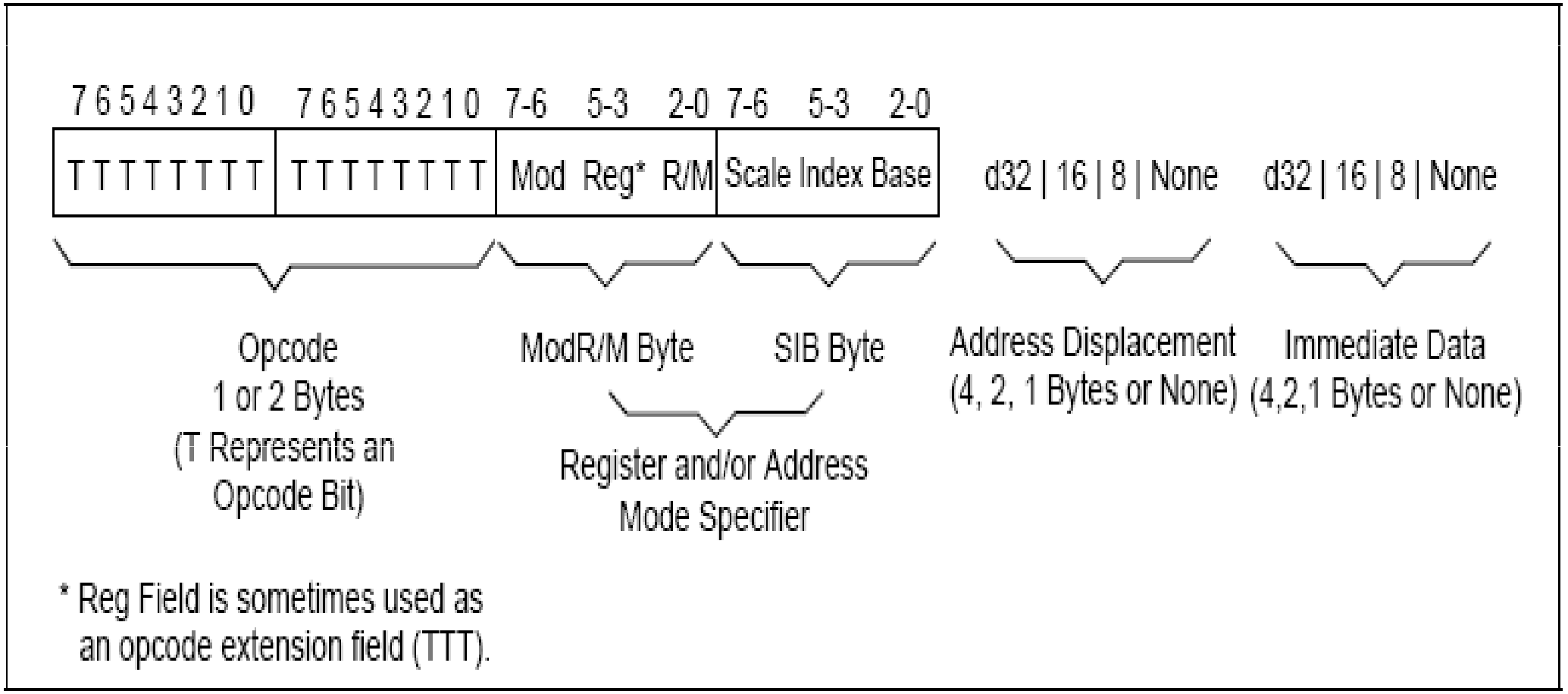


Figure B-1. General Machine Instruction Format

Mettre les différentes OPCODE de jmp, jne, je, etc au lieu de ADC

Exemple 1 : ADC addition avec retenue

ADC—Add with Carry

Opcode	Instruction	Description
14 *ib*	ADC AL,*imm8*	Add with carry *imm8* to AL
15 *iw*	ADC AX,*imm16*	Add with carry *imm16* to AX
15 *id*	ADC EAX,*imm32*	Add with carry *imm32* to EAX
80 /2 *ib*	ADC *r/m8,imm8*	Add with carry *imm8* to *r/m8*
81 /2 *iw*	ADC *r/m16,imm16*	Add with carry *imm16* to *r/m16*
81 /2 *id*	ADC *r/m32,imm32*	Add with CF *imm32* to *r/m32*
83 /2 *ib*	ADC *r/m16,imm8*	Add with CF sign-extended *imm8* to *r/m16*
83 /2 *ib*	ADC *r/m32,imm8*	Add with CF sign-extended *imm8* into *r/m32*
10 /*r*	ADC *r/m8,r8*	Add with carry byte register to *r/m8*
11 /*r*	ADC *r/m16,r16*	Add with carry *r16* to *r/m16*
11 /*r*	ADC *r/m32,r32*	Add with CF *r32* to *r/m32*
12 /*r*	ADC *r8,r/m8*	Add with carry *r/m8* to byte register
13 /*r*	ADC *r16,r/m16*	Add with carry *r/m16* to *r16*
13 /*r*	ADC *r32,r/m32*	Add with CF *r/m32* to *r32*

Exemple 1 : ADC addition avec retenue

- L'instruction ADC additionne le premier opérande (nommé destination) avec le second opérande (nommé source) et le drapeau de retenue. ADC met son résultat dans l'opérande destination (le premier en lisant de gauche à droite)
- Les deux opérandes ne peuvent être simultanément des adresses en mémoire.
- L'opérande source peut-être une valeur immédiate.
- Le drapeau CF (carry flag) représente une retenue d'un addition précédente.

Exemple 1 : ADC

Operation

DEST ← DEST + SRC + CF;

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Exemple 2 : CMP compare 2 opérandes

CMP—Compare Two Operands

Opcode	Instruction	Description
3C *ib*	CMP AL, *imm8*	Compare *imm8* with AL
3D *iw*	CMP AX, *imm16*	Compare *imm16* with AX
3D *id*	CMP EAX, *imm32*	Compare *imm32* with EAX
80 /7 *ib*	CMP *r/m8, imm8*	Compare *imm8* with *r/m8*
81 /7 *iw*	CMP *r/m16, imm16*	Compare *imm16* with *r/m16*
81 /7 *id*	CMP *r/m32,imm32*	Compare *imm32* with *r/m32*
83 /7 *ib*	CMP *r/m16,imm8*	Compare *imm8* with *r/m16*
83 /7 *ib*	CMP *r/m32,imm8*	Compare *imm8* with *r/m32*
38 /*r*	CMP *r/m8,r8*	Compare *r8* with *r/m8*
39 /*r*	CMP *r/m16,r16*	Compare *r16* with *r/m16*
39 /*r*	CMP *r/m32,r32*	Compare *r32* with *r/m32*
3A /*r*	CMP *r8,r/m8*	Compare *r/m8* with *r8*
3B /*r*	CMP *r16,r/m16*	Compare *r/m16* with *r16*
3B /*r*	CMP *r32,r/m32*	Compare *r/m32* with *r32*

Détailler CMP r/m32, r32 et CMP r/m32, i32

Exemple 2 : CMP compare 2 opérandes

Description

This instruction compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

Exemple 2 : CMP compare 2 opérandes

- L'instruction CMP compare deux opérandes : le premier avec le second, puis positionne les drapeaux du registre d'état conformément au résultat.
- La comparaison est effectuée en soustrayant le second opérande du premier. CMP positionne les drapeaux du registre d'état de la même façon que l'instruction SUB (soustraction)
- Lorsqu'une valeur immédiate est utilisée comme seconde opérande, alors elle est étendue en conservant son signe à la taille du premier opérande.

Les Macros en NASM

```
%macro   nomMacro    n          ;;; n est le nombre de paramêtres
     OPCODE1   <operandes>
     ...
     OPCODEp   <operandes>
%endmacro
          ;;; Dans le corps de la macro,
          ;;; %1 dénote le premier argument en ligne   arg1
          ;;; %n référence argn
APPEL DE LA MACRO :
          nomMacro arg1 , ... , argn
```

Donnez la définition des macros pour les appels de fonctions avec arguments

RISC versus CISC

- C'est dans les années 80 que la notion de RISC est apparue. Elle consiste à minimiser le nombre d'instructions et à les simplifier!
- L'autre tendance (CISC) est de rajouter autant d'instructions que possible au processeur.
- Dans les années 80, des chercheurs d'**IBM,** sous la direction de **John Cocke** se sont convaincus qu'un ensemble réduit d'instructions " rapides/efficaces" valait mieux qu'un plus grand ensemble augmenté d'instructions plus lentes et moins efficaces.

RISC versus CISC

- Les processus RISC sont moins chers car moins complexes et ne comporterent que des instructions très rapides toutes codées sur un même nombre d'octets. Elles sont plus simples et plus homogènes.
- Les professeurs David Patterson (Berkeley) et John Hennessy (Stanford) sont ceux qui apposèrent les noms de CISC et RISC aux deux philosophies.

RISC	CISC
Peu d'instructions (~100)	Beaucoup d'instructions (~ 1000)
Instructions rapides: 1 cycle = 1 instruction	Des instructions parfois lentes (> 1 cycle)
Instructions toutes codées sur 4 octets	Instructions codées sur 1 à 15 octets
Une base + 1 déplacement pour l'adressage	Format complexe car plusieurs formats utilisés pour l'adressage mémoire
Rapide: Opérations arithmétiques et logiques sur les registres uniquement	Opération arithmétiques et logiques à la fois sur des registres et de la mémoire
Moins souple: Contraintes d'implémentation: séquence d'instructions interdites	Implémentation transparente
Efficace: Seuls des tests, dont le résultat va dans des registres, sont utilisés lors des branchements conditionnels	Des drapeaux sont positionnés et utilisés lors des branchements conditionels
Passage des paramètres: Utilisation uniquement des registres pour les arguments des fonctions ainsi que pour l'adresse de retour	Passage des paramètres: Utilisation intensive de la pile pour les arguments et pour l'adresse de retour

RISC versus CISC

- Dans les années 80 la communauté scientifique a longuement débattu de l'avantage de l'une et l'autre philosophie.
- Dix ans plus tard, il est apparu que l'une et l'autre avait des avantages. Ainsi les processus RISC ont tendance à devenir de + en + CISC et vice-versa.
- La technologie CISC domine le marché des ordinateurs de bureau et des ordinateurs portables. La technologie RISC domine le marché des microprocesseurs embarqués.